Dokumentierung von Nim-Spiel Programmierung

Der gesamte benötigte Code zur Ausführung des Nim-Spiels in einer LVP-Umgebung

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Random;
import java.util.stream.IntStream;

class Move {
    final int row, number;
    static Move of(int row, int number) {
        return new Move(row, number);
    }
    private Move(int row, int number) {
        if (row < 0 || number < 1) throw new IllegalArgumentExcept
        this.row = row;
        this.number = number;
    }
    public String toString() {
        return "(" + row + ", " + number + ")";
    }
}

interface NimGame {
    static boolean isWinning(int... numbers) {
        return Arrays.stream(numbers).reduce(0, (i,j) -> i ^ j) !=

    }
    NimGame play(Move... moves);
    boolean isGameOver();
    String toString();
```

```
}

class Nim implements NimGame {
    private Random r = new Random();
    int[] rows;
    public static Nim of(int... rows) {
        if(rows.length > 5 || !(Arrays.stream(rows).allMatch(n ->
            throw new IllegalArgumentException("Die Anzahl an Reih
        }
         return new Nim(rows);
    }
    private Nim(int... rows) {
        assert rows.length >= 1;
        assert Arrays.stream(rows).allMatch(n -> n >= 0);
        this.rows = Arrays.copyOf(rows, rows.length);
    }
    private Nim play(Move m) {
        assert !isGameOver();
        assert m.row < rows.length && m.number <= rows[m.row];
        Nim nim = Nim.of(rows);
        nim.rows[m.row] -= m.number;
        return nim;
    }
    public Nim play(Move... moves) {
        Nim nim = this;
        for(Move m : moves) nim = nim.play(m);
        return nim;
    }
    public boolean isGameOver() {
        return Arrays.stream(rows).allMatch(n -> n == 0);
    }
    @ Override
    public int hashCode() {
        int sum = 0;
       for(int value : this.rows){
             sum += Math.pow(value, 2);
        }
        return sum ;
    }
    public boolean equals (Object other){
```

```
        if(other == null) return false;
        if(other == this) return true;
        if(this.getClass() != other.getClass()) return false;
        Nim that = (Nim)other;
        return this.hashCode() == that.hashCode();
    }
    @ Override
    public String toString() {
        NimView.show(this);
        String s = "";
        for(int n : rows) s += "\n" + "I ".repeat(n);
        return s;
    }
}

class NimView  {
        Nim nim ;
       static Turtle turtle = new Turtle();


       NimView(Nim nim){
            this.nim = nim;

        }

        static void show(Nim nam){
            turtle.reset();
            int x , y = 80;
            String s = "";

            for(int i = 0; i < nam.rows.length; i++){
                x = 50;
                for(int j = 0; j < nam.rows[i]; j++){
                 turtle.moveTo(x,y).left(90).lineWidth(15).forwarc
                     x = x + 80;
                    turtle.right(90).moveTo(x, y);

                } s = "\n";
                y += 100;
                turtle.moveTo(100,y);
            }
```

```
        }


}
```

Explizite Erklärung der Aufgabenteile

Implementierung der Methode Equals

Mit der Implementierung dieser Methode wollen wir prüfen können, ob zwei Nim-Objekte gleich sind. Die Methode nimmt als Parameter ein anderes Objekt und soll so aufgebaut sein, dass sie den Spielverlauf beachtet. Beispielweise ist Nim.of(1,3,5) == Nim.of(5,3,1) == Nim.of(1,3,5,0). Zum erreichen dieses Ziel prüft die Methode zuerst, ***ob das Parameter null ist***. wenn es der Fall ist, dann fertig, denn null kann nie mit einem bereits existierenden Objekt gleich sein. Wenn nicht der Fall , ***dann machen wir weiter mit der Überprüfung der Reflexivität, also ob beide Objekte eigentlich ein einziges Objekt sind***. und wenn ja, dann sind sie selbstverständlich gleich.Wenn nein, dann prüft die Methode , ***ob beide Objekte zur gleichen Klasse gehören***. Wenn nein , dann können sie auf keinen Fall gleich sein , wenn ja dann ***machen wir weiter mit Casting, um zu allen Eigenschaften des Objekts greifen zu können***, und letzentlich wird Überprüft,***ob beide Objekte den gleichen Hashcode haben***, wenn es der Fall ist, dann sind sie gleich , wenn nicht der Fall dann sind sie nicht gleich. Das ganze sieht dann wie folgt aus :

```
@Override
    public boolean equals (Object other){

        if(other == null) return false;
        if(other == this) return true;
        if(this.getClass() != other.getClass()) return false;
        Nim that = (Nim)other;
        return this.hashCode() == that.hashCode();
    }
```

Implementierung der Methode Hashcode

Jetzt wollen wir für jedes Nim-Objekt so eine eindeutige Zahl selbst geben, die es als „ID“ dient. Das schaffen wir durch die Implementierung eigener Hashcode-Methode; und wir wollen nämlich so verfahren , dass diese Methode den Spielverlauf auch beachtet. Das heißt die Reihenfolge der Reihe muss nicht berücksichtigt werden und das Hinzufügen einer Reihe, die 0 Hölzchen enthält , soll kein Problem darstellen. zum Beispiel Nim.of(1,3,5) , Nim.of(3,5,1) und Nim.of(5,1, 3,0) sollen alle den selben Hashcode haben. Diese ist nämlich möglich, ***indem wir jede Zahl des rows-Arrays zum Quadrat erheben und dann summieren***.Dadurch wird weder die Reihenfolge der Zahlen noch das Hinzufügen einer 0-Zahl den Hashcode beeinflussen denn `1*1 + 3*3 + 5*5 == 3*3 + 5*5 +1*1 == 5*5 + 1*1 + 3*3 + 0*0 == 35`). Der Code sieht dann wie folgt aus :

```
@ Override
   public int hashCode() {
       int sum = 0;
      for(int value : this.rows){
            sum += Math.pow(value, 2);
       }
       return sum ;
   }
```

Exception werfen

Das Spielfeld sollte so aufgebaut sein , dass es maximal 5 Reihen und 7 Hölzchen pro Reihe enthalten soll. Diese beschränkung schaffen wir, durch das Werfen einer Exception in der statistischen Methode of(Methode, die beim Erzeugen von Objekten hilft, da der Konstruktor privat ist).

```
public static Nim of(int... rows) {
    if(rows.length > 5 ||  !(Arrays.stream(rows).allMatch(n ->
        throw new IllegalArgumentException("Die Anzahl an Reih
    }
     return new Nim(rows);
}
```

Der Bedingungsteil `rows.length > 5` prüft,ob die Anzahl an Reihen im Array <= 5 ist

und `!(Arrays.stream(rows).allMatch(n -> n <= 7))` prüft,ob die Anzahl an Hölzchen pro Reihe <= 7 ist. Stream ist tatsächlich ein Werkzeug, das uns ermöglicht, Sammlungen von Daten (zB. Arrays oder Listen) auf eine funktionale und flüssige Weise zu verarbeiten,ohne dass man explizit Schleifen schreiben muss. ***AllMatch(n -> n…)*** wiederum ist eine Methode von stream , die zum Prüfen einer bestimmten Bedingung verwendet wird. Ist eine von beiden Bedingungen beim Erzeugen von Objekten verletzt , so wird eine Exception mit folgender Nachricht geworfen :`"Die Anzahl an Reihen soll <= 5 und die Anzahl an Hölzchen pro Reihe soll <= 7 sein "`

die Klasse NimView

NimView soll für die Darstellung im Browser zuständig sein, das schaffen wir indem NimView Nim als Objektvariable und eine statistische Methode `show` enthält. Die Methode show wiederum soll mithilfe des Turtle-Clerks die aktuelle Spielsituation mit den Hölzchen im Browser anzeigen. Gehen wir einbisschen tiefer auf darauf ein.

Verwendung von Nim innerhalb der NimView-Klasse

Hier wurde eine Variable namens nim als Objektvariable deklariert und initialisiert wie folgt :

```
class NimView  {
    Nim nim ;
    static Turtle turtle = new Turtle();


    NimView(Nim nim){
    this.nim = nim;

    }
```

Die Show-Methode

`show` nimmt als Parameter eine Variable von Typ Nim. Diese ist notwendig, denn der Aufruf von show in der toString-Methode von Nim soll auf jeden Spielzug in der Jshell den aktuellen Spielstand im Browser parallel und gleichzeitig aktualisieren können. Das Verfahren ist , sodass show die Turtle als statistische Variable verwendet, und folgende Anweisungen und Methode auf die Turlte verwendet

Befehl	**Bedeutung**
`reset()`	Lösche Zeichenfläche, gehe zurück in Bildmitte. ist notwendig bei der Aktualisierung des Spielfeldes
`moveTo(x,y)`	bewege dich an der Stelle x,y
`forward(double distance)`	Bewege dich um *distance* vorwärts
`right(double degrees)`	Drehe dich um die Gradzahl *degrees* nach rechts
`left(double degrees)`	Drehe dich um die Gradzahl *degrees* nach links
`color(int rgb)`	Setze Stiftfarbe auf den kodierten RGB-Farbwert *rgb*
`lineWidth(double width)`	Setze Stiftbreite auf *width*

`int x , y = 80; x = 50;` Mit der Deklaration und Initialisierung von x und y, wird die initiale Position der Schildkröte bestimmt.

```
    static void show(Nim nam){
        turtle.reset();
        int x , y = 80;
        String s = "";

        for(int i = 0; i < nam.rows.length; i++){
            x = 50;
            for(int j = 0; j < nam.rows[i]; j++){
                turtle.moveTo(x,y).left(90).lineWidth(15).forward(
                x = x + 60;
                turtle.right(90).moveTo(x, y);

            }
             s = "
";
```

```
            y += 100;
            turtle.moveTo(100,y);
        }
    }
```

`turtle.moveTo(x,y).left(90).lineWidth(15).forward(50);` Diese Anweisung hilft, ein Hölzchen darzustellen. Mit `x = x + 60;` wird der Abstand zwischen Hölzchen gleicher Reihe erstellt. Mit `turtle.right(90).moveTo(x, y);` wird die Schildkröte nach rechts umgedreht und Ihre Position aktualisiert. Die innere For-Schleife hilft insgesamt dazu die Hölzchen gleicher Reihe nebeneinander darzustellen, deswegen soll der Zähler j bis nam.rows[i] durchlaufen. Die äußere For-Schleife wiederum wird benutzt um die Reihen darzustellen.deswegen läuft i bis die Länge des Arrays - 1 durchlaufen. Die Anweisung new line ist auch nötig , um die Zeile zu wechseln , nachdem die Hölzchen einer Reihe schon vollständig dargestellt wurden. `y += 100;` kümmert sich um den Abstand zwischen Reihen. turtle.moveTo(x,y); aktualisiert die Position des ersten Hölzchens jeder Reihe. der ganze Code sieht wie folgt aus :

```
    static void show(Nim nam){
        turtle.reset();
        int x , y = 80;
        String s = "";

        for(int i = 0; i < nam.rows.length; i++){
            x = 50;
            for(int j = 0; j < nam.rows[i]; j++){
                turtle.moveTo(x,y).left(90).lineWidth(15).forward(
                x = x + 60;
                turtle.right(90).moveTo(x, y);

            }
             s = "
";
            y += 100;
            turtle.moveTo(100,y);
        }
    }
```

Das Ganze lässt sich in der toString-Methode von Nim durch die Anweisung `NimView.show(this);` . ***Da die Methode static ist kann man direkt den Klassenname Nimview benutzen***. das Wort **this** wiederum **steht für das Nim-Objekt, an dem es gerade gearbeitet wird**. Und der ganze Aufruf sorgt für **eine rechtzeitige Darstellung im Browser**

```
Lassen wir uns jetzt spielen 😉. Dafür erzeugen wir zuerst ein Ob
```

```
Nim nim = Nim.of(2,4,5,6,7)
nim = nim.play(Move.of(4,4));
```

Das Ergebnis sieht so aus :